Using learning analytics in the Amazonas: understanding students' behaviour in introductory programming

Filipe Dwan Pereira, Elaine H. T. Oliveira, David Oliveira, Alexandra I. Cristea, Leandro Galvão, Samuel Fonseca, Armando Toda and Seiji Isotani

Filipe Pereira is auxiliary professor of Computer Science (CS), Federal University of Roraima. Elaine Oliveira is associate professor of CS, Federal University of Amazonas (UFAM). David Oliveira is associate professor of CS, UFAM. Alexandra Cristea is Professor, Deputy Head at the CS Department, Durham University. Leandro Galvão is associate professor and the coordinator of the programme of CS, UFAM. Samuel Fonseca is a computer engineering research fellow, UFAM. Armando Toda is a PhD student, University of São Paulo (USP). Seiji Isotani is Professor, USP. Address for correspondence: Elaine Oliveira. Email: elaine@icomp.ufam.edu.br.

Abstract

Tools for automatic grading programming assignments, also known as Online Judges, have been widely used to support computer science (CS) courses. Nevertheless, few studies have used these tools to acquire and analyse interaction data to better understand students' performance and behaviours, often due to data availability or inadequate granularity. To address this problem, we propose an Online Judge called CodeBench, which allows for fine-grained data collection of student interactions, at the level of, e.g., keystrokes, number of submissions, and grades. We deployed CodeBench for three years (2016-2018) and collected data from 2058 students from 16 introductory computer science (CS1) courses, on which we have carried out fine-grained learning analytics, towards early detection of effective/ineffective behaviours regarding learning CS concepts. Results extract clear behavioural classes of CS1 students, significantly differentiated both semantically and statistically, enabling us to better explain how student behaviours during programming have influenced learning outcomes. Finally, we also identify behaviours that can guide novice students to improve their learning performance, which can be used for interventions. We believe this work is a step forward towards enhancing Online Judges and helping teachers and students improve their CS1 teaching/learning practices.

Introduction

Many undergraduate programs in STEM (Science, Technology, Engineering and Mathematics) have an introductory programming course, generally known as CS1 (Computer Science 1), as a mandatory component of their curriculum. However, literature reports high failure rates in CS1 (Bennedsen & Caspersen, 2007; Watson & Li, 2014; Ihantola et al., 2015b; Costa et al., 2017; Lacave et al., 2018; Pereira et al., 2019c). More specifically, non-Computer Science major students may experience difficulties at higher levels, as they may not usually have typical intrinsic motivation (Norman & Adams, 2015; Santana & Bittencourt, 2018; Echeverría et al., 2019).

Practitioner Notes

What is already known about this topic:

- Studies suggest that one-third of CS1 students end up failing.
- Learning to program takes a lot of practice and feedback is highly desirable.
- Student activity in Online Judges can be used to predict their outcome, but studies are few.
- Data for such studies is often proprietary or of inadequate granularity.

What this paper adds:

- A new Online Judge system, CodeBench, which allows for fine-grained learning analysis of student behaviour for CS1.
- Employing learning analytics to identify early effective behaviours for novice students and, for the first time in our knowledge, how these behaviours can be useful for ineffective students.
- A novel classification of students into effective, average and ineffective, based on their behaviour, which shows both semantic and significant statistical differences.
- A clear indication that student behaviour during programming influences learning outcomes for CS1.
- A proposal, design, and implementation of a large scale, longitudinal study of student behaviour in CS1.

Implications for practice and/or policy:

- Students may need to reflect on their behaviour and self-regulate.
- Instructors may propose specific strategies and guidance based on early ineffective behaviours.
- Instructors, institutes and other educational stakeholders have a powerful learning analytics tool to understand student behaviour and patterns.

A common agreement from computing education research (CEdR) is that programming students need practice and quick feedback on the correctness of their code (Ihantola et al., 2015b; Robins, 2019; Dwan et al. 2017; Pereira et al., 2019c). However, instructors usually face large and heterogeneous classes, which makes individualised support almost impractical (Per-Arne et al., 2016).

At the same time, Massive Open Online Courses (MOOCs) are proliferating, supporting thousands of students. Yet, they provide little feedback, functionality or personalisation to their heterogeneous learners, increasing the dropout rate and demotivation regarding CS learning (Shah, 2018). Additionally, research in the areas of Intelligent Tutoring Systems, Adaptive Educational Hypermedia, and AI in Education have shown, albeit mainly on small scale teaching experiments, that personalisation is useful (Kulik & Fletcher, 2016).

To improve the way students learn to program and to tackle the high dropout rate, instructors from the Federal University of Amazonas, Brazil, have developed an Online Judge from scratch, called *CodeBench*, which automatically evaluates and feeds back on CS1 assignments. Our home-made system combines the large-scale approach of the popular MOOC formula with the flexibility given by an in-house system, allowing unprecedented research depth and amenability, based on Learning Analytics (LA). Running since 2016, CodeBench has collected interaction data from 2058 non-CS major students, across six semesters, from 16 different classes every year. It collects fine granularity data, while students attempt to solve assignments and exams, such as keystrokes in the embedded Integrated Development Environment (IDE), mouse clicks, lesson material, submissions, etc.

Thus, in this work, we model this fine-grained data using learning analytics methods, to identify early effective programming behaviours and how they could be useful to guide ineffective students. As a step towards understanding CS1 student behaviours, our goal is to answer the following two research questions:

RQ1: *How can effective and ineffective behaviours of CS1 students be detected early, using data from an Online Judge system?*

RQ2: *Which effective behaviours of novice students can be useful to guide students with ineffective behaviours?*

Definition of effectiveness, ineffectiveness and resilience

In this paper, we call "effective students" those who make progress in learning to program, typically leading to successful outcomes (Robins, 2019). Conversely, "ineffective students" are those who do not make progress or require excessive effort, typically leading to unsuccessful outcomes (Robins, 2019). Finally, we use the term "resilience" to refer to the students who struggle to edit and submit code more than the median of attempts.

Related Work

Despite the considerable number of studies about novice and expert learners from introductory programming classes, a very recent work, Robins (2019), explains that we know very little about effective and ineffective behaviours of novice students in CS1 classes, especially how effective behaviours can be identified and whether they can be used to improve the learning process of the ineffective learners.

In this sense, Edwards et al. (2009) conducted an initial study, involving 1,101 students from three different computer science courses (CS1, CS2, and CS3). The authors inspected students' time and code-size data and found that students who achieve better grades start and finish assignments earlier than students with worse grades. In addition, successful novice programmers moderately wrote more program code. Although this study has analysed an extensive sample, they inspected just a few features (procrastination and changes in the codes). In our paper, we employ hence a much larger set of 16 features (Table 2), as well as a larger number of students (Table 1) with a new method to detect early effective behaviours.

Data collected from Online Judges can be useful for formative assessment, prediction of student outcomes and early identification of students at risk (Herodotou et al. 2019). Recently, these kinds of analyses have been arousing the interest of researchers. For example, Estey & Coady (2016); Otero et al. (2016); Dwan et al. (2017); Pereira et al. (2019a); Pereira et al. (2019b); Van Leeuwen et al. (2019) use code metrics, such as number of submissions, time spent programming, temporal patterns, number of syntactic errors, a.o., to estimate students outcomes, using varied machine learning and data mining techniques.

Still, Hosseini et al. (2014) performed a formative assessment of programming students by evaluating how 101 learners develop their code over time in order to explore problem solving paths. Rivers & Koedinger (2017) investigated a small dataset of 15 programming students collected from a what they called intelligent teaching assistant for programming with the aim of generating automatic hints. Both works are much smaller scale than ours; they support, like our work, a fine-grained data analysis, as crucial for a better understanding of programming behaviours.

Jadud (2006) proposed an algorithm called *Error Quotient (EQ)*, which uses snapshots of compilation to quantify the student errors while they are programming. The EQ algorithm receives as input a pair of compilation events and assigns to them a penalty, if both events ended with error. The penalty could vary, e.g., whether or not the error of both compilation events was the same. Watson et al. (2013) extended EQ with an algorithm to compute the *Watwin Score (WS)*, which scores compilation pairs, by additionally considering the problem-solving time.

Considering learning analytics in the context of CS1 classes to model students' behaviour, studies have explored the compilation errors (Jadud, 2006; Watson et al., 2013), how students deal with deadlines (Spacco et al. ,2013; Vihavainen, 2013; Auvinen, 2015), how many attempts they need to solve the problems (Ahadi et al., 2016), how they use hints in a web-based system (Estey & Coady, 2016), which are their frequencies of submission and unique attempts (Munson & Zitovsky, 2018), how much effort they put into code (static analysis) writing (Otero et al., 2016), and which is their typing pattern and keystroke latency when programming (Leinonen, 2016). All these studies analyse the relationship between students' code metrics and performance.

Here, instead, we start with all fine-grained code metrics (features) in CodeBench, proposed our self-devised ones, as a basis for selecting the best. Hence, one important contribution of this work is to show, through our case study, how important it is to measure, collect, analyse and report the fine-grained data proposed by the CodeBench, and how this helps stakeholders to have an early understanding of students' behaviours in programming classes.

Educational Context

At the Federal University of Amazonas, about 640 STEM undergraduate students enrol every year in CS1, distributed over sixteen classes, 11 held in the first term and 5 in the second. Figure 1 shows the evolution of pass rates from 2010 to 2018.

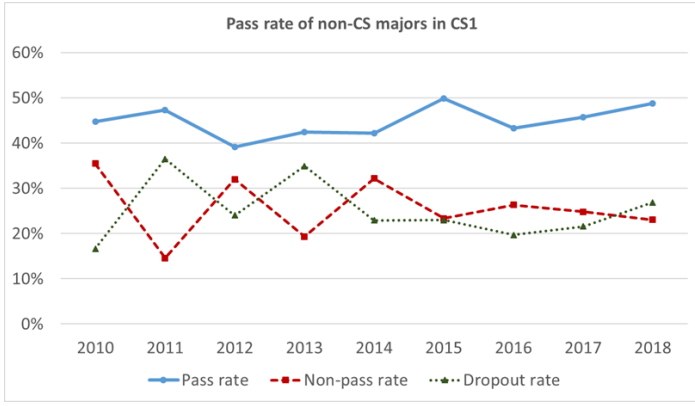

Figure 1: Evolution of pass rate in CS1 classes

In public Brazilian universities, students must go through a new selection process if they want to move from one undergraduate program to another. Thus, the dropout rate of CS1 is partially explained by the dissatisfaction of some students with the program itself.

Since 2015, the content is divided into seven modules, each consisting of four two-hour sessions: (1) variables; (2) conditionals; (3) nested conditionals; (4) *while* loops; (5) vectors; (6) *for* loops; and (7) matrices. Each module follows this sequence of activities: a lecture class, two practical classes, and a partial exam. Python is adopted as the base programming language.

Since CS1 content gradually increases in complexity, the final grade is a weighted average of the partial exams, where more advanced modules have higher weights. The final grade is determined based on 7 partial exams (summative assessments), 7 mandatory programming assignments (formative assessments), and one final exam. Partial exams contribute with increasing weights (6.1% to 18.2%) to the final grade, whereas all assignments have the same weight (≈1.3%). For this study, we normalised the final grades in the range from 0 to 1.

Methods

Instrument

CodeBench[1] (Figure 3) is an Online Judge developed from scratch by one of the authors. It evaluates students' codes using a "dynamic analysis approach", as defined by the taxonomy proposed by Ullah et al. (2018), i.e. students submit their program to the platform, and the output is compared against a set of test cases, manually provided by the instructor.

[1] codebench.icomp.ufam.edu.br

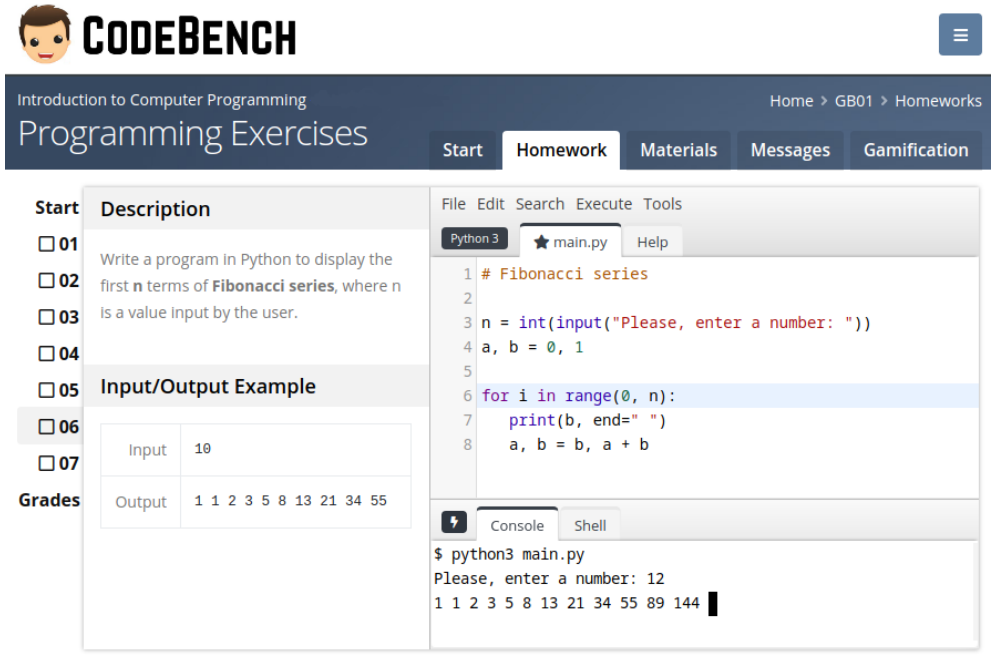

Figure 3: Screenshot of the student interface of CodeBench, showing a programming assignment composed of 7 questions (left), the description of question #6 (middle), the built-in IDE (right), and the Python console (right bottom)

Data Collection

CodeBench provides its own code editor (IDE), designed to be simple and novice-friendly, as shown in Figure 3. All actions performed by students in the IDE (e.g., keystrokes, submissions, code pasting, mouse clicks, tab or window transitions, etc.) are timestamped and recorded in a log file on the server side. Figure 4 shows an example of the log, from where we can extract variables to measure learner behaviour.

```
1  27/5/2016@8:34:42:871:focus
2  27/5/2016@8:34:43:475:change:{"from":{"line":0,"ch":0},"to":{"line":0,"ch":0},"text":["p"],"removed":[""],"origin":"+input"}
3  27/5/2016@8:34:43:615:change:{"from":{"line":0,"ch":1},"to":{"line":0,"ch":1},"text":["r"],"removed":[""],"origin":"+input"}
4  27/5/2016@8:34:43:677:change:{"from":{"line":0,"ch":2},"to":{"line":0,"ch":2},"text":["i"],"removed":[""],"origin":"+input"}
5  27/5/2016@8:34:43:811:change:{"from":{"line":0,"ch":3},"to":{"line":0,"ch":3},"text":["n"],"removed":[""],"origin":"+input"}
6  27/5/2016@8:34:43:884:change:{"from":{"line":0,"ch":4},"to":{"line":0,"ch":4},"text":["t"],"removed":[""],"origin":"+input"}
7  27/5/2016@8:34:43:884:change:{"from":{"line":0,"ch":5},"to":{"line":0,"ch":5},"text":["()"],"removed":[""],"origin":"+input"}
```

Figure 4: Logs collected from CodeBench when the learner was writing a “print” command

We have analysed data from 2016-1 to 2018-2 (2016-1 means the first term of 2016), with emphasis on the *first four weeks*, since our goal is to detect early effective and ineffective behaviour. After gathering data from six academic terms (semesters), we had 2058 students in total, which we call *consolidation data*[2]. As it is difficult to compare data from different runs, we applied a *z*-score z_i to each feature (briefly explained in the next section), i.e., assigning zero to the mean $\bar{x}$ and replacing values x_i with the amount of standard deviations $\sigma(x_i)$ from this mean for any value other than the mean, as follows:

$$z_i = \frac{x_i - \bar{x}}{\sigma(x_i)}$$

Table 1 presents descriptive statistics, differentiated by term, on the number of students and submission attempts for the programming assignments and exam exercises.

[2] Our dataset can be found on codebench.icomp.ufam.edu.br/dataset/

Table 1: Number of instances in CodeBench's data set (by term)

	2016-1	*2016-2*	*2017-1*	*2017-2*	*2018-1*	*2018-2*	*Total*
Students	535	176	481	190	486	190	2058
Programming assignments	675	447	1278	556	1550	893	5399
Exam exercises	128	110	153	93	180	107	771
Submission attempts	154163	38933	119370	27613	148775	46765	535619

Features Extraction and Selection

We extracted useful information from two sources in CodeBench: IDE raw logs and students' source codes. We defined which features to be extracted, firstly based on previous work, to which we added self-devised features, based on discussions with three of the instructors (who also authored this paper). The features suggested by the instructors were mainly related to coding activity, time spent in the IDE and inappropriate use of copy&paste.

We computed several features, such as the proportion of copy&paste events, number of executions between submissions, number of deleted characters, program metrics (number of cycles, cyclomatic complexity, number of variables, number of comments, number of non-comment lines, etc.), among others. In order to reduce the feature space, we analysed the pairwise Spearman's rho correlation among all the features, since they were not normally distributed (Shapiro-Wilk $\gg$0.05). Hinkle et al. (1988) claims that a correlation ≥ 0.9 (absolute value) is strong. As such, we removed features with correlation bellow -0.9 and above 0.9 with other features. In the case of a high correlation between a pair of features, we opted to remove the one with a lower correlation with the final grade, as we intend to find features related to effective and ineffective behaviour.

Table 2 describes each remaining feature along with its categorisation, according to a taxonomy of 'useful information' derivable from IDE data, proposed by Carter et al. (2019), where *Count* represents features that can be extracted by counting events in raw log files or source codes, *Math* represents features that need a mathematical formula to be computed, and *Algo* represents those features that need an algorithm applied in the raw data to extract useful information.

Table 2: Features (programming behaviour) used to cluster CS1 students

Feature (programming behaviour)	**Description**	**Type**
procrastination	Time (delay) between first code edit and programming assignment deadline, multiplied by -1 (Carter et al., 2019; Edwards et al., 2009)	Count
amountOfChange	In a pair of consecutive submissions of the same problem, *amountOfChange* refers to the number of lines added or changed between the two submissions (Carter et al., 2019; Edwards et al., 2009)	Count
event_activity	A binary value which equals 0 when a student solves a problem within less than a number[3] of log lines (events), or 1 otherwise (self-devised)	Algo
attempts	Average number of submission attempts per problem (regardless whether correct or not) (Ahadi et al., 2016)	Math
lloc	Average number of logical lines for each submitted code (Otero et al., 2016). Imports, comments, and blank lines were not counted	Math
systemAccess	Number of student logins up to the end of the fourth week (adapted from Pereira et al. (2019b)	Count
firstExamGrade	Student grade in the first exam, taken in the fourth week of the course (self-devised)	Count

[3] One standard deviation minus the median of the numbers of *events* for a problem

events[4]	Number of log lines on attempting to solve problems. To illustrate, each time the student presses a button in the embedded IDE of CodeBench, this event is stored as a line in a log file (adapted from Leinonen et al. (2016) and Castro-Wunsch et al. (2017)	Count
correctness	Number of problems correctly solved from programming assignments before the first exam (Pereira et al., 2019a)	Count
copyPaste	Proportion between pasted characters ('ctrl+V') and characters typed (Pereira et a., 2019a)	Math
syntaxError	Ratio between the number of submissions with syntax error[5] and the number of attempts (Estey & Coady, 2016; Pereira et a., 2019a)	Math
ideUsage	Total time (in minutes) spent by students solving problems in the embedded IDE (it removes inactive time – more than 5 minutes without typing) -Adapted from Pereira et al. (2019b).	Algo
keystrokeLatency	Keystroke average latency of the students when typing in the embedded IDE (we also removed downtime) – adapted from Leinonen et al. (2016)	Algo
errorQuotient	Compute a score based on the number of code errors and repeated errors (Jadud, 2006)	Algo
watWinScore	An extension of *errorQuotient* which considers the problem-solving time (Watson et al., 2013)	Algo
countVar	Number of variables in the source code (Carter et al., 2019)	Count

Figure 5 presents pairwise Spearman's rho correlations among the features in Table 2. It shows that *watWinScore*, *procrastination*, *copyPaste*, *syntaxError*, and *errorQuotient* are positively correlated. However, they are negatively correlated with other features: *amountOfChange*, *attempts*, *lloc*, *systemAccess*, *firstExamGrade*, *events*, *correctness*, *ideUsage*, *eventActivity*, and *keystrokeLatency*. This suggests that high values of features of the first group might correspond to negative/undesirable behaviour (e.g., high *syntaxError* is an undesirable behaviour, which should be improved). This observation is further used in the clustering analysis.

Evaluative Factors

As effectiveness and ineffectiveness are psychological constructs, we derived a set of evaluative factors to perform an operational definition of these concepts. For each student, we determined the number of solved and unsolved questions, considering the number of attempts of code submission to CodeBench correction. At the same time, for each programming question, we calculated the median of the number of attempts. We state here that students made *few attempts* if they made less than the median of attempts, and *many attempts*, otherwise. Moreover, if students perform many attempts, they are called *resilient*. From these assumptions, we defined five evaluative factors to measure effective or ineffective behaviours, as follows:

- Non-attempt ratio: $noAttempt = \frac{\#\ non-attempted\ questions}{\#\ questions}$
- Unsuccessful without resilience ratio: $unsucNoRes = \frac{\#\ unsolved\ questions\ after\ few\ attempts}{\#\ questions}$
- Successful without resilience ratio: $sucNoRes = \frac{\#\ solved\ questions\ after\ few\ attempts}{\#\ questions}$
- Unsuccessful with resilience ratio: $unsucRes = \frac{\#\ unsolved\ questions\ after\ many\ attempts}{\#\ questions}$
- Successful with resilience ratio: $sucRes = \frac{\#\ solved\ questions\ after\ many\ attempts}{\#\ questions}$

[4] Castro-Wunsch et al. (2017) called this feature *'number of steps'*. However, unlike them, we averaged it.

[5] *SyntaxError* is a common and generic exception in Python.

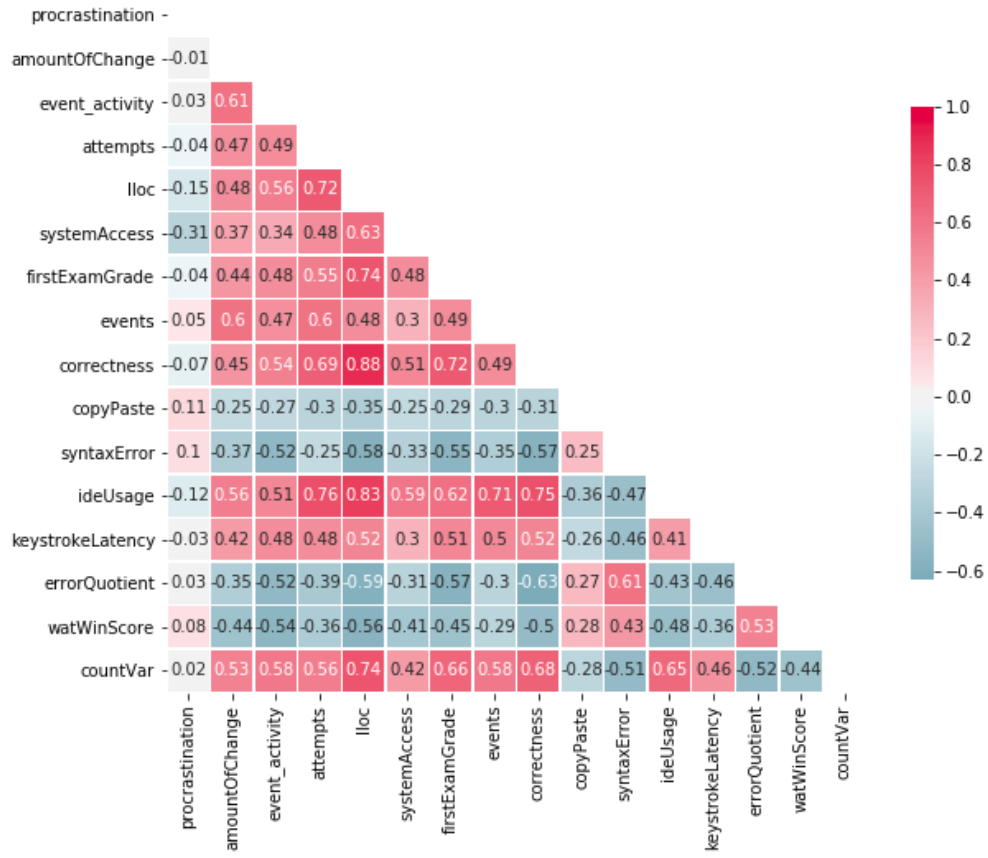

Figure 5: Pairwise Spearman's rho correlation between remained features

Note that the sum of these five factors is equal to 1. In addition, we defined another two evaluative factors, based on how successfully learners solve programming assignments, as follows:

- Effective attempt ratio: $effectAttRate = \frac{\#\ solved\ questions}{\#\ attempted\ questions}$
- Effective general ratio: $effectGenRate = \frac{\#\ solved\ questions}{\#\ questions}$

Here we hypothesised that an effective behaviour is related to low values of *noAttempt* and *unsucNoRes*. Similarly, it is related to high values of *sucNoRes*, *sucRes*, *effectAttRate*, *effectGenRate*. On the other hand, ineffectiveness is the opposite. To check if our hypotheses are reasonable, we measured the correlation of the evaluative factors with the final grades. Figure 6 shows the pairwise Spearman rho correlation[6] among evaluative factors and final grade.

Regarding *unsucRes*, a high value may look like a sign of ineffectiveness. When students try a lot, even not achieving a correct solution, they show resilience, which is an important characteristic for programmers (Pereira et al., 2019c). In addition, Online Judges based on dynamic analysis have limitations and sometimes they may be unfair, since they perform a string comparison between student solution output and the expected output. Consequently, if the student misses a space or a break line in the output (presentation error), the solution will be considered as wrong, even if it is logically correct. Thus, to confirm this assumption, we have analysed the correlation between

[6] We applied Spearman rho correlation because the evaluative factors are normally distributed

unsucRes and *finalGrade*. We found a positive value (r_s = .65) and, hence, we state that a high *unsucRes* is related to effectiveness.

Notice that *sucNoRes* has a weak positive correlation with *finalGrade* (r_s = 0.05). Although this outcome also sounds unexpected, Ahadi et al. (2016) showed that there are cases in which solving correctly the programming problem is irrelevant, provided that the learner achieves a threshold of attempts (resilience). Still, there are also problems that are important to solve correctly (success), but students are expected to have done so with more than a specific number of attempts (with resilience). Moreover, at the beginning of the course, students are learning how to deal with the nuances of CodeBench and it is common to make naive mistakes (as previously explained), provoking an increase on the number of attempts even for students who end up passing.

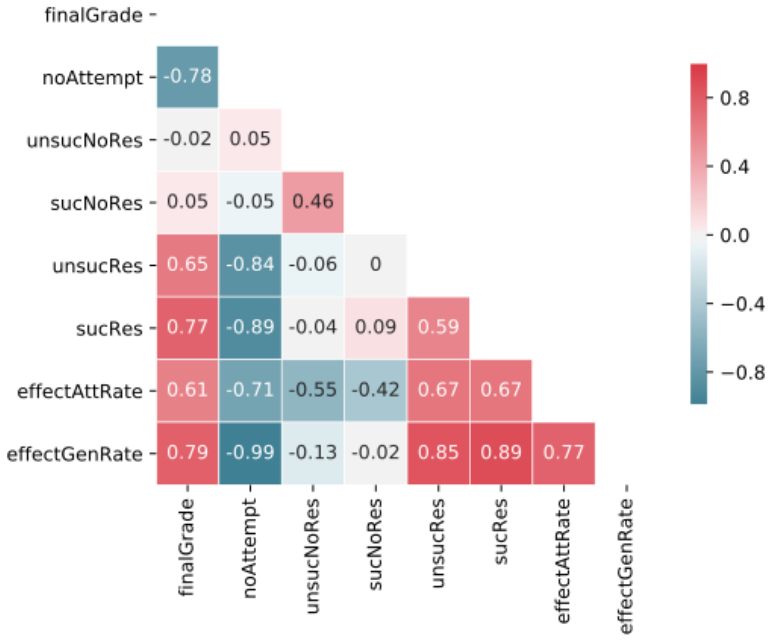

Figure 6: Pairwise Spearman's rho correlation between evaluative factors

Thus, different from our prior assumption, effectiveness and ineffectiveness without resilience are not correlated with the final grade (Figure 6). Hence, we kept as evaluative factors only the features with Spearman's rho correlation with the final grade above .6, which were: *noAttempt*, *unsucRes*, *sucRes*, *effectAttRate*, *effectGenRate*, and obviously the *finalGrade*.

Clustering and Association Rule algorithms

Clustering algorithms can uncover hidden patterns in a complex dataset and many works (Antonenko et al. 2018; Shi et al. 2019; Shi & Cristea 2018; Dutt et al. 2015) used these unsupervised learning methods to analyse new relationships on educational data. Still, students' behaviour is heterogenous and, as effective and ineffective programming indicators need to be found, these would be expected to have different values for different student subpopulations. Hence, we cluster students based on the students' logs, in order to inspect the patterns of programming behaviours in each student cluster and how these behaviours reflect on the evaluative factors. To do so, we used the well-known *k*-means algorithm (MacQueen, 1967). This algorithm clusters n observations within a predetermined number of k clusters, where each observation belongs to the nearest group mean. Thus, each observation is closer to its own cluster centroid[7]. Henceforth, we have used the mean silhouette coefficient (Rousseeuw, 1987) of observations to choose the most appropriate number of clusters for our data, as this method can be applied to analyse the distance between every pair of clusters.

[7] Represented by the mean of the observations within the cluster

Furthermore, to strengthen and validate our conclusions and triangulate results, Association Rule Mining (ARM) is used to identify groups within a given dataset, based on the support (frequency) of items (Agrawal et al., 1993). The effectiveness of ARM can be evaluated through different measures; in this paper, we opted to use *confidence* and *lift*, since those are some of the most used in the literature (Huang et al. 2017).

Results and Discussion

Analysing Consolidation Data

To tackle the research questions, we modelled students' programming behaviour using the features presented in Table 2. These served as observations of the k-means clustering method. From the complete three-year data set, we have used data from the first four weeks of the course, as we aimed to detect effective behaviour early on. Previous studies support that it is possible to draw patterns using fine-grained data from the first weeks of introductory programming courses (Munson & Zitovsky, 2018; Pereira et al., 2019a; Estey & Coady, 2016).

We inspected the relationship between the *k*-means clusters and the effectiveness or ineffectiveness (based on evaluative factors). The convergence of *k*-means was achieved in the 10^{th} iteration with $k=3$ as the best value, and 44.94% of the students were assigned to Cluster A, 29.93% to Cluster B, and 25.12% to Cluster C. We applied the non-parametric Mann-Whitney U test for pairwise cluster comparison of features, to inspect the impact of each feature, individually, in the cluster formation. The results indicate a statistical difference (even with Bonferroni correction: $p \ll 0.05/3$) between features in different clusters, except for the *errorQuotient* between Clusters A and B.

Furthermore, in terms of evaluative factors, Figure 7 shows that Cluster A performs better than Cluster B, which performs better than Cluster C, and, by transitivity, Cluster A performs better than Cluster C. Indeed, this pattern is kept in terms of all evaluative factors presented in the methods section. To check the statistical significance between these evaluative factors, we conducted a pairwise comparison between each cluster, as shown in Table 3. Apart from *unsucRes* for *Cluster A vs Cluster B* only, Cluster A outperforms Cluster B, which outperforms Cluster C for all evaluative factors, even with Bonferroni correction ($p \ll 0.05/3$).

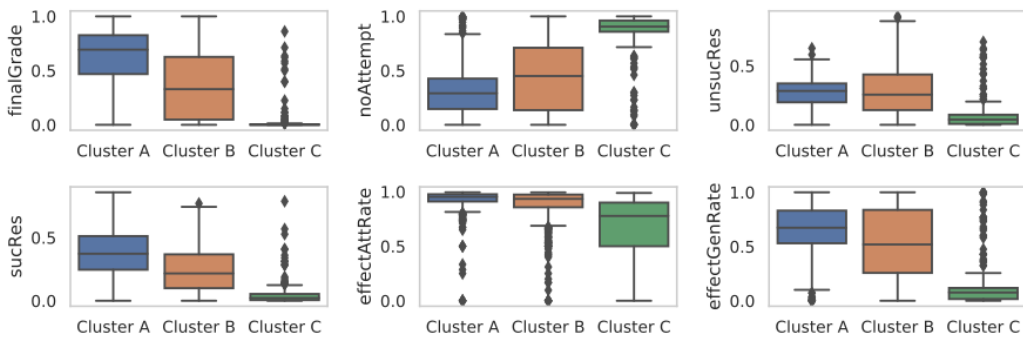

Figure 7: Evaluative factors distribution of each cluster

Based on Figure 7 and pairwise Mann-Whitney tests of the evaluative factors (Table 3), we can label each cluster, as follows: (i) Cluster A contains the *effective* students, (ii) Cluster B comprises the *average* students, and (iii) Cluster C has the *ineffective* ones. Moreover, Figure 8 presents the centroids of each cluster, showing the general programming behaviour of each group. From Figure 8, we can see which early programming behaviours (represented by the *k*-means centroids) are leading factors for effectiveness or ineffectiveness by adopted clusters.

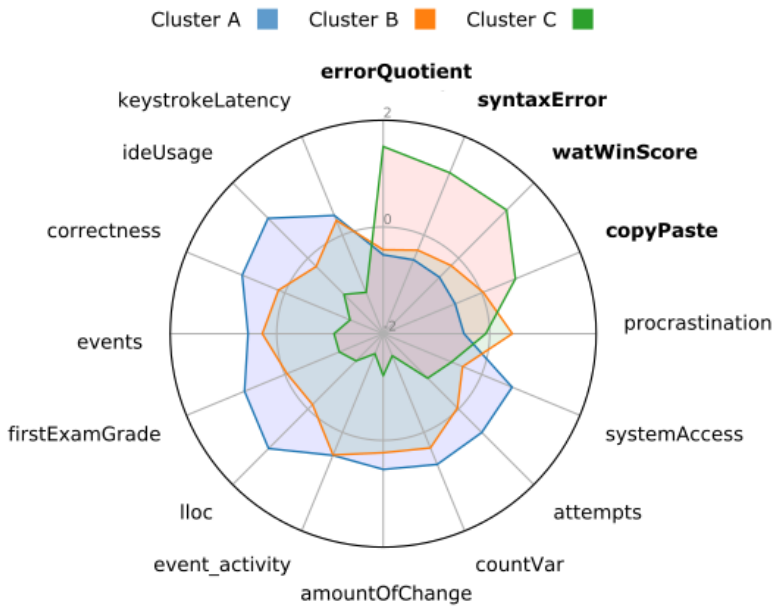

Figure 8: Programming profile of students in each cluster. We have marked with bold and grouped together the features with a reverse scale for a better visualisation.

Comparing Student Clusters

First, comparing effective with ineffective students, we can observe that the effective ones deal with errors better, since they have a lower *errorQuotient*, and they take less time to correct mistakes, as they have a lower *watWinScore* and, hence, they tend to not get stuck too much time with, for example, the same errors. Besides, effective learners have a higher *amountOfChange* between submissions, which might explain why these learners can fix errors faster. Coupled with that, ineffective learners are more affected with *syntaxError*, a generic and recurring exception in Python that may be difficult to fix for CS1 students. With that in mind, instructors might be notified, or even the learners can be made aware of the risk of having many erroneous submission pairs, especially if the error is the generic Python *SyntaxError*. Moreover, a clear sign of concern is represented by a student with a huge difference in timestamps between a pair of submissions with errors (mainly with the same errors), coupled with a small amount of change.

Equally important, effective students have more *attempts*, *lloc*, *systemAccess*, *events*, *ideUsage*, and *countVar*, as they spend more time programming, and they submit more problems to the Online Judge. This clearly shows that these learners are more engaged with the course. Furthermore, effective students tend to code faster (higher *keystrokeLatency*). They also solve more problems (higher *correctness*) which is one possible reason why they achieve a better grade in the first exam (higher *firstExamGrade*). Finally, these students tend to manage their time better, as they procrastinate less. Still, when they solve problems, they do so with more *events*. If students solve a problem with too few *events* (lower *eventActivity*) they are copying and pasting code already created, which may or may not have been developed by themselves. In this sense, we can also see from Figure 8 that ineffective students tend to copy and paste more, which is not a desirable behaviour for introductory students at the very beginning of the course (first four weeks in our case).

To compare the average students with the two other groups, we should consider the effect sizes (Cohen, 2013) from Tables 3. We can see for all features a medium to large ($r > 0.4$) degree[8] to which a sample from Cluster A (effective students) has stochastic dominance compared with the other sample from Cluster C (ineffective students). However, we can see a lower degree ($r < 0.3$) to which the null hypothesis (sample from the same group) is false comparing Cluster A versus Cluster

[8] Mangiafico (2016) states that r in between 0.10 and 0.30 is considered small, while r greater than 0.30 and lower than 0.50 is considered medium, and $r > 0.5$ is a large effect.

B and comparing Cluster B versus Cluster C, which shows that Cluster B has nuances from both groups, i.e., students from Cluster B might have effective and ineffective behaviours.

Table 3: Pairwise cluster comparisons of evaluative factors

		finalGrade	noAttempt	unsucRes	sucRes	effectAttRate	effectGenRate
Cluster A vs Cluster B	r	0.1405	0.0317	0.0000	0.1440	0.0168	0.0320
	Asy. Sig (2)	0.0000	0.0000	0.9020	0.0000	0.0000	0.0000
Cluster A vs Cluster C	r	0.5807	0.6626	0.6079	0.6694	0.4021	0.6602
	Asy. Sig (2)	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Cluster B vs Cluster C	r	0.1405	0.0317	0.0000	0.1440	0.0168	0.0320
	Asy. Sig (2)	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Association rules analysis

A total of 83 rules[9] were found (0.1 < support < 0.6, confidence > 0.1 and lift = 6.0850. Figure 9 presents the stronger rules, sorted by their confidence (strongest rules have a higher contrast). The figure shows how different clusters (A, B, C) are more influenced by some features than by others.

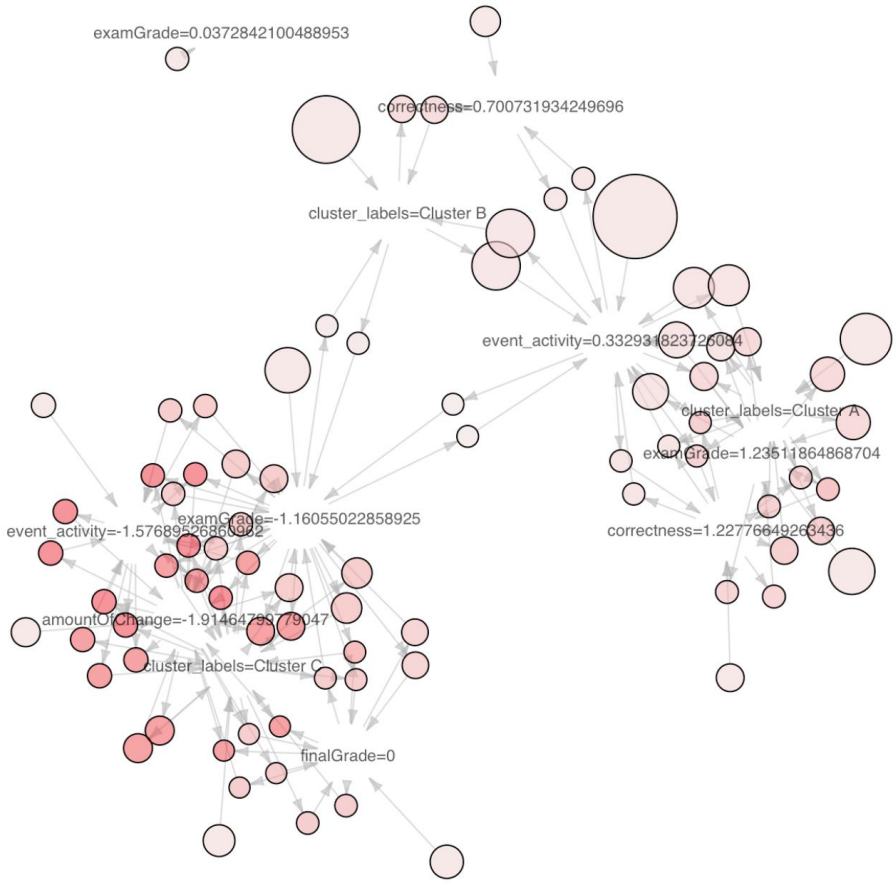

Figure 9: Association Rules representation

[9] The full set of association rules can be accessed on shorturl.at/czEOX

In general, the rules confirmed what we found for effective and ineffective behaviour using clustering, e.g., the *amountOfChange*, *eventActivity* and *examGrade* are highly associated (confidence >= 0.8, lift > 3) with the behaviour of non-passing students. As a new finding, we observed that low *eventActivity* with a small *amountOfChange* is related to ineffective students (lift = 6.1, confidence = 0.8, rule 55). On the other hand, effective learners have solved more problems (higher correctness) with higher *eventActivity* (lift = 3.59, confidence = 0.56, rule 64). By finding these new associations, alongside the similarities within our clusters, we can infer that these rules might be useful (due to the high lift) as predictors to identify at-risk students.

Finally, knowing which behaviour can be effective for CS1 learners may help towards self-regulation and awareness of what kind of attitude can be dangerous and detrimental to their performance. An easily implementable option is to share this information as a report to instructors, to provide them with tools to motivate students. For instance, we have shown that resilient students (higher *sucRes* and *unsucRes* – Cluster A), who do not easily give up trying to solve a problem, usually perform better and end up passing. Instructors can then encourage weaker students to be resilient, as a path towards better performance. However, resilience alone is not enough. There are more factors involved in effectiveness, such as knowing how to fix, analyse or debug compilation errors, managing well their time to solve the assignments, and practising to develop skills - such as a higher accuracy of problem-solving, and so forth.

Pedagogical Implications

The origins of Online Judges trace back to programming competitions, whose intent was to test programming ability, instead of building it. More recently, Online Judges are used as a self-learning tool, in parallel to regular courses. Thus, this work impacts on a wide variety of stakeholders, such as developers, teachers, students. Programming is a hands-on activity, which, however, requires a great amount of feedback. Such feedback is precious in educational terms, and *early feedback* is vital, if behavioural changes are desired. However, it is not scalable to large class sizes and growing number of students, as in the case studied here, of the Amazonas. Instead, with an approach such as ours, effective and ineffective behaviours in introductory programming can be *automatically* identified early on, and encouraged or discouraged, respectively. On the other hand, some measured behaviours may not be straightforward or wise to automatically action upon. In such situations, instructors can receive alerts of which students are at risk of low performance and why, so they can reflect on possible causes of the observed ineffective behaviours. For example, explicitly inefficient student behaviours (such as procrastination, copying/pasting of big chunks of code, and low IDE usage time) can be automatically tracked by an Online Judge. Students could themselves be directly alerted, as a first port of call, on how far their behaviour is from the higher-performance students within their group, as well as in average, and as a result be invited to change their study strategy.

As another example, copy&paste behaviour may be due to plagiarism, or simply due to writing code separately and only pasting it in the system when ‘ready’. Nevertheless, writing code directly in the system is very informative in the analysis and can lead to much more refined feedback to students. Thus some initial notification on undesired behaviour can be useful for students to be given a chance to change their own behaviour to better reflect their knowledge status. In addition, these alerts can be sent to instructors and tutors, so they can reinforce, offline, the need for changing study behaviour.

Other behaviours, such as a high keystroke latency and a small amount of change in code may be an observed consequence of non-observed actions. In this case, this information should be reported only to instructors, who, in turn, should plan activities in order to address the cause of such behaviours. For example, they can assign extra exercises that target debugging, misconceptions, or code patterns.

Furthermore, identified effective behaviours can be brought to the attention of instructors, effective, but also ineffective students – with care about non-disclosure of personal information. For example, late students can be warned when a certain number of students complete assignments or spend more time coding in the IDE than they do. Here, again, group membership can inform the feedback, and weak students comparing themselves with the best amongst their group, as opposed to the best in class, which may be impossible for them to ‘beat’. Additionally, the Online Judge can notify

instructors about which students need extra help, in good time before any deadlines, allowing for *proactive instead of reactive pedagogical interventions*. Finally, it is worth noting that, whilst correlation or even association rules per se do not imply causation, the two-pronged triangulation approach used is providing more evidence towards prediction power. Moreover, undesirable behaviours may need to be addressed in some cases, even if they may not directly cause ineffectiveness – as in the example of the copy&paste case.

Conclusions, Limitations and Future Works

CS1 classes usually have high heterogeneity among students. This was clear in this work, showing variations in the patterns of student behaviours, resulting in three different clusters. We further supported the findings via statistical differences in learning outcomes, programming behaviour and evaluative factors between these clusters. Importantly, our analyses showed which early (based on only the first four weeks) programming behaviours potentially indicate effectiveness or ineffectiveness in learning.

This result can support decision making of students as well as instructor intervention, such as designing specific guidance for a struggling group of students, proposing new and challenging exercises for effective students, and personalising exercises, according to different student needs. Furthermore, knowing which behaviour can be effective might help students to improve their self-regulation and awareness of what kind of programming behaviour can be dangerous or beneficial to their performance.

Among the limitations of this study is the data collected from a single institution, which may affect the generalisation of the results. However, considering data was collected from CS1 courses from several years and students from different majors, this limitation might be reduced. As future work, we envision to analyse how the data-driven approach used in this paper can model students who begin the course with successful behaviours but end up with failure behaviours and grades.

References

Agrawal, R., Imieliński, T., Swami, A., Agrawal, R., Imieliński, T., Swami, A., 1993. Mining association rules between sets of between sets of items in large databases. URL: https://dl.acm.org/doi/10.1145/170035.170072, doi:10.1145/170035.170072

Ahadi, A., Vihavainen, A., Lister, R., 2016. On the number of attempts students made on some online programming exercises during semester and their subsequent performance on final exam questions. ACM conference on Innovation and Technology in Computer Science Education, 218–223.

Antonenko, P.D., Toy, S., Niederhauser, D.S. 2012. Using Cluster Analysis for Data Mining in Educational Technology Research. Educational Technology Research and Development. 60, 3 (Jun. 2012), 383–398.

Auvinen, T., 2015. Harmful study habits in online learning environments with automatic assessment, in: Learning and Teaching in Computing and Engineering (LaTiCE), 2015 International Conference on, IEEE. pp. 50–57.

Bennedsen, J. & Caspersen, M.E., 2019. Failure rates in introductory programming: 12 years later. ACM Inroads 10, 2 (April 2019), 30-36.

Brin, S., Motwani, R., Ullman, J.D., Tsur, S., 1997. Dynamic itemset counting and implication rules for market basket data. ACM SIG-MOD Record 26, 255–264.

Carter, A., Hundhausen, C., Olivares, D., 2019. Leveraging the Integrated Development Environment for Learning Analytics. In S. Fincher & A. Robins (Eds.), *The Cambridge Handbook of Computing Education Research* (Cambridge Handbooks in Psychology, pp. 679-706). Cambridge: Cambridge University Press.

Castro-Wunsch, K., Ahadi, A., Petersen, A., 2017. Evaluating neural networks as a method for identifying students in need of assistance. Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, 111–116.

Cohen, J., 2013. Statistical power analysis for the behavioral sciences. Routledge.

Costa, E.B., Fonseca, B., Santana, M.A., de Araújo, F.F., Rego, J., 2017. Evaluating the effectiveness of educational data mining techniques for early prediction of students' academic failure in introductory programming courses. Computers in Human Behavior 73, 247– 256.

Dutt, A., Aghabozrgi, S., Ismail, M. A. B., Mahroeian, H. (2015). Clustering algorithms applied in educational data mining. International Journal of Information and Electronics Engineering, 5(2), 112.

Dwan, F., Oliveira, E., Fernandes, D., 2017. Predição de zona de aprendizagem de alunos de introdução à programação em ambientes de correção automática de código, in: Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação SBIE), p. 1507.

Echeverría, L, Cobos, R, Machuca, L, Claros, I., 2017. Using collaborative learning scenarios to teach programming to non-CS majors. Comput Appl Eng Educ.; 25: 719–731.

Edwards, S.H., Snyder, J., Pérez-Quiñones, M.A., Allevato, A., Kim, D., Tretola, B., 2009. Comparing effective and ineffective behaviors of student programmers, in: Proceedings of the fifth international workshop on Computing education research workshop, ACM. pp. 3– 14.

Estey, A. & Coady, Y., 2016. Can interaction patterns with supplemental study tools predict outcomes in CS1? Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '16, 236–241.

Herodotou, C., Hlosta, M., Boroowa, A., Rienties, B., Zdrahal, Z., Mangafa, C. (2019). Empowering online teachers through predictive learning analytics. British Journal of Educational Technology, 50(6), 3064-3079.

Hinkle, D.E., Wiersma, W., Jurs, S.G., 1988. Applied statistics for the behavioral sciences. Houghton Mifflin Boston.

Hosseini, Roya and Vihavainen, Arto and Brusilovsky, Peter (2014) Exploring Problem Solving Paths in a Java Programming Course. In: Psychology of Programming Interest Group Conference, PPIG 2014.

Huang, H., Tornero-Velez, R., Barzyk, T.M., 2017. Associations between socio-demographic characteristics and chemical concentrations contributing to cumulative exposures in the United States. Journal of Exposure Science and Environmental Epidemiology 27, 544–550. URL: https://www., doi:10.1038/jes.2017.15.

Ihantola, P., Vihavainen, A., Ahadi, A., Butler, M., Börstler, J., Edwards, S.H., Isohanni, E., Korhonen, A., Petersen, A., Rivers, K., Rubio, M., Sheard, J., Skupas, B., Spacco, J., Szabo, C., Toll, D., 2015b. Educational data mining and learning analytics in programming: Literature review and case studies. ACM. Proceedings of the 2015 ITiCSE on Working Group Reports, 41–63.

Jadud, M.C., 2006. Methods and tools for exploring novice compilation behaviour. Proceedings of the second international workshop on Computing education research, 73–84.

Kulik, J.A. & Fletcher, J., 2016. Effectiveness of intelligent tutoring systems: a meta-analytic review. Review of Educational Research 86, 42–78.

Lacave, C., Molina, A.I., Cruz-Lemus, J.A., 2018. Learning analytics to identify dropout factors of computer science studies through bayesian networks. Behaviour & Information Technology 37, 993– 1007.

Leinonen, J., Longi, K., Klami, A., Vihavainen, A., 2016. Automatic inference of programming performance and experience from typing patterns. Proceedings of the 47th ACM Technical Symposium on Computing Science Education, 132–137.

MacQueen, J., 1967. Some methods for classification and analysis of multivariate observations. Proceedings of the fifth Berkeley symposium on mathematical statistics and probability 1, 281–297.

Mangiafico, S.S., 2016. Summary and analysis of extension program evaluation in r. Rutgers Cooperative Extension: New Brunswick, NJ, USA.

Munson, J.P. & Zitovsky, J.P., 2018. Models for early identification of struggling novice programmers, in: Proceedings of the 49th ACM Technical Symposium on Computer Science Education, ACM. pp. 699–704.

Norman, V. T. & Adams, J. C., 2015. Improving Non-CS Major Performance in CS1. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 558-562.

Otero, J., Junco, L., Suarez, R., Palacios, A., Couso, I., Sanchez, L., 2016. Finding informative code metrics under uncertainty for predicting the pass rate of online courses. Information Sciences 373, 42–56.

Per-Arne, A., Kråkevik, C., Goodwin, M., Yazidi, A., 2016. Adaptive task assignment in online learning environments. ACM. Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics, 5.

Pereira, F.D., Oliveira, E., Cristea, A., Fernandes, D., Silva, L., Aguiar, G., Alamri, A., Alshehri, M., 2019a. Early dropout prediction for programming courses supported by online judges, in: International Conference on Artificial Intelligence in Education, Springer. pp. 67–72.

Pereira, F.D., Oliveira, E.H., Fernandes, D., Cristea, A., 2019b. Early performance prediction for CS1 course students using a combination of machine learning and an evolutionary algorithm, in: 2019 IEEE 19th International Conference on Advanced Learning Technologies (ICALT), IEEE. pp. 183–184.

Pereira, F., Oliveira, E., Fernandes, D., Junior, H., Carvalho, L. S. G. 2019c. Otimização e automação da predição precoce do desempenho de alunos que utilizam juízes online: uma abordagem com algoritmo genético. In Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE) (Vol. 30, No. 1, p. 1451).

Robins, A. V. "Novice programmers and introductory programming." In S. Fincher & A. Robins (Eds.), The Cambridge Handbook of Computing Education Research (Cambridge Handbooks in Psychology, pp. 327-376). Cambridge: Cambridge University Press.

Rivers, K. & Koedinger, K. R. (2017). Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. International Journal of Artificial Intelligence in Education, 27(1), 37-64.

Rousseeuw, Peter J. "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis." Journal of computational and applied mathematics 20 (1987): 53-65.

Santana, B. L. & Bittencourt, R. A., 2018. Increasing Motivation of CS1 Non-Majors through an Approach Contextualized by Games and Media. IEEE Frontiers in Education Conference (FIE), San Jose, CA, USA, pp. 1-9.

Shah, D., 2018. By the numbers: Moocs in 2018. https://www.classcentral.com/report/mooc-stats-2018/. Accessed: 2019-10-29.

Shi, L., Cristea, A. I., Toda, A. M., Oliveira, W. (2019). Revealing the Hidden Patterns: A Comparative Study on Profiling Subpopulations of MOOC Students. In A. Siarheyeva, C. Barry, M. Lang, H. Linger, & C. Schneider (Eds.), Information Systems Development: Information Systems Beyond 2020 (ISD2019 Proceedings). Toulon, France: ISEN Yncréa Méditerranée.

Shi, Lei & Cristea, A. I.. "In-depth Exploration of Engagement Patterns in MOOCs." International Conference on Web Information Systems Engineering. Springer, Cham, 2018.

Spacco, J., Fossati, D., Stamper, J., Rivers, K., 2013. Towards improving programming habits to create better computer science course outcomes, in: Proceedings of the 18th ACM conference on Innovation and technology in computer science education, ACM. pp. 243–248.

Ullah, Z, Lajis, A, Jamjoom, M, Altalhi, A, Al-Ghamdi, A, Saleem, F., 2018. The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. *Comput Appl Eng Educ*.; 26: 2328–2341.

van Leeuwen, A., Bos, N., van Ravenswaaij, H., van Oostenrijk, J. (2019). The role of temporal patterns in students' behavior for predicting course performance: A comparison of two blended learning courses. British Journal of Educational Technology, 50(2), 921-933.

Vihavainen, A., 2013. Predicting students' performance in an introductory programming course using data from students' own programming process. Proceedings - 2013 IEEE 13th International Conference on Advanced Learning Technologies, ICALT 2013, 498–499.

Wasik, S., Antczak, M., Badura, J., Laskowski, A., Sternal, T., 2018. A survey on online judge systems and their applications. ACM Computing Surveys (CSUR) 51, 3.

Watson, C. & Li, F.W., 2014. Failure rates in introductory programming revisited. Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education, 39–44.

Watson, C., Li, F.W.B., Godwin, J.L., 2013. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. 2013 IEEE 13th International Conference on Advanced Learning Technologies, 319–323.